



SHADOW-OBJECT INTERFACE BETWEEN FORTRAN 95 AND C++

Mark G. Gray, Randy M. Roberts, and Tom M. Evans

WE ARE OBJECT-ORIENTED ENTHUSIASTS. WHILE OUR PREFERRED LANGUAGE IS C++, MANY OF OUR COLLEAGUES PREFER FORTRAN 95. PREVIOUSLY IN *COMPUTERS IN PHYSICS*, WE EXPLAINED HOW TO GET THE MOST OUT OF FORTRAN 95

by using what we called an *object-based style*.¹ Since then, C++ has acquired a good standard; quality compilers are widely available, and *expression-template* technology has improved its performance to Fortran 95 levels.² Mixed-language programming on large projects is now common, and we want to interface these two powerful languages without losing their power by reducing ourselves to least-common-denominator features.

Ideally, we want to automatically interface C++ and Fortran 95 code, with either language playing the role of `main` and with user-defined types from one language available in the other. We envision something like Figure 1, where the behavior of an object in language *x* is exported to language *y* by a flat physical interface consisting of intrinsic types or simple (1D) arrays of those types, and the object's identity and state is exported to *y* by a shadow object, which presents a logical interface. In the following sections, we describe how to physically interface C++ and Fortran 95, how to logically interface them, and

what this means for their in scientific programming.

Physically interfacing C++ and Fortran 95

To physically interface C++ and Fortran 95 we must:

- *Name unmanage*: ensure that procedure names are visible and sensible across the language interface.
- *Flatten*: reduce procedure arguments to the common set of built-in types available in both languages
- *Initialize*: ensure that the proper code initialization takes place in both languages

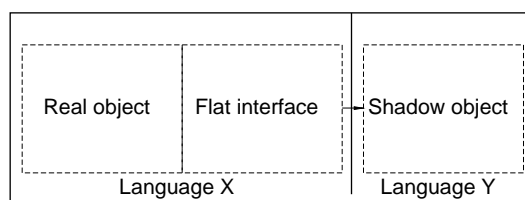


Figure 1. The interface between *x* and *y*. An object in language *x* is physically interfaced to language *y* through a flat interface. A shadow object in *y* uses the physical interface to provide a logical interface.

Name unmangling. Both C++ and Fortran 95 mangle procedure names. C++ mangles all procedure names and Fortran 95 mangles all module procedure names, both in a compiler-dependent fashion. C++ provides the `extern "C"` construct to interface with C code, which (almost) does not mangle names. Similarly, Fortran 95 supports the external-procedure construct that (almost) does not mangle names. The physical interface in C++ must therefore consist of a set of `extern "C"` procedures. Likewise, the physical interface in Fortran 95 must consist of external procedures.

Actually, even the names of `extern "C"` procedures in C++ and external procedure in Fortran 95 are mangled; the procedure names entered into the symbol table might have their case modified and an underscore appended or prepended according to compiler-dependent rules. The convention a particular compiler uses is easy to discern: simply compile the code and use `nm` to examine the symbol table. Various schemes^{4,5} to automatically generate the appropriate names are available. For simplicity here, we manually write the interface routines with the appropriate case and underscores for one particular Fortran 95 name-mangling scheme.

Flattening. Interfacing through nonmember func-

Mark G. Gray is a physicist in the Transport Methods Group at Los Alamos National Laboratory.

//Author: Please briefly list your research interests, education (with degrees, majors, & schools), a career highlight, and your professional memberships.// Contact him at Los Alamos National Lab, Los Alamos, NM 87545; gray@lanl.gov; <http://laws.lanl.gov:80/~gray>.

Randy M. Roberts is a physicist in the Transport Methods Group. His research interests include radiation/neutron transport, thermodynamics, fluid mechanics, electromagnetic wave propagation, underwater acoustics, and surface physics. He has a BS in physics from the University of Maryland and a PhD in physics from the University of Texas. He is a member of the American Physical Society. Contact him at rsqrd@lanl.gov; <http://www.xdiv.lanl.gov/~rsqrd>.

Tom M. Evans is a physicist in the Transport Methods Group. His research interests include hybrid deterministic/Monte Carlo transport, radiative transfer, charged-particle physics, and radiation detection. He has a BS in physics and astronomy from Haverford, and an MS in health physics and a PhD in nuclear engineering, both from the Georgia Institute of Technology. He is a member of the American Association of Physicists in Medicine, the American Institute of Physics, and the American Nuclear Society. Contact him at tme@lanl.gov; <http://laws.lanl.gov:80XTM/tme>.

tions (in C++) or external procedures (in Fortran 95) is also necessary for interface flattening. In addition, flattening requires that the procedure parameter list contain only the common types intrinsic to both languages, or simple (1D) arrays of those types. Of course, the internal representation of those types must match across the language barrier for the interpretation to be successful. Fortunately, almost all modern compilers use the IEEE standard representation for its intrinsic types.

For most platforms, the following intrinsic types match between Fortran 95 and C++:

<i>type</i>	<i>C++</i>	<i>Fortran 95</i>
Integer	long	integer
Real	double	real
Character	char	character

Because Fortran 95 passes all arguments by reference, we restrict all the arguments on the C++ side to references (or pointers) of these types.

Although this list seems rather restrictive for the modern language user, much useful work has been done in Fortran 77 using only these types. Further, other intrinsic types could be used (such as Fortran `logical`), with the appropriate translation. As we shall see, the logical interfacing techniques to come

will permit the use of user-defined types constructed from these basic types.

Initialization. Initialization of static and `const` objects is a major obstacle when interfacing C++ and Fortran 95. A code with a C++ `main` initializes `const` and `static` data before `main` is executed. (The C++ standard stipulates that this condition is not binding; function-scope objects can be instantiated when first encountered); a code with a Fortran 95 `main` will probably not initialize this data. This problem is especially acute for libraries that contain `const` or `static` variables with namespace or class scope.⁶

We can solve the initialization problem most elegantly by using shared libraries. ELF based shared-library implementations automatically initialize themselves correctly.⁷ We take advantage of this by compiling all foreign language code into a shared library. The shared library takes care of the initialization. This works on Linux, Sun, and SGI platforms, but does not work on IBM platforms (an alternative approach would be to use a, perhaps dummy, C++ `main` routine).

```
// Declaration of foo (the real class)

#ifndef FOO_H
#define FOO_H

class foo                                // C++ real class
{
public:
    foo();                                // build foo
    int get();                            // return member data value
    void set(int j);                      // set member data
    ~foo();                               // destroy foo
private:
    int i;                               // member data
};

#endif
```

Figure 2. C++ class header. We wish to create a flat physical interface to this class's member functions so they can be invoked from Fortran 95, and a logical interface to this class's objects so they can be shadowed in Fortran 95.

Café Dubois

POOMA II

The Café is buzzing today with news from Scott Haney about a release by Los Alamos National Laboratory's Advanced Computing Laboratory (ACL) of several C++ software packages. These packages are available free for noncommercial use and can be downloaded from <http://www.acl.lanl.gov/software>.

Former *Computers in Physics* readers might recall the "Scientific Programming" department in last year's September-October issue about the ACL's Parallel Object-Oriented Methods and Applications (POOMA) framework. A new release, POOMA 2.0, is now available.

Scott is very enthusiastic about the new ideas in POOMA II. POOMA II has a flexible array class that supports a plug-in *engine* architecture to achieve representation independence. It includes a powerful system for specifying and combining domains to construct views of arrays. These views are arrays, so they can be used anywhere an array is expected. Using a novel *expression-engine* abstraction, array expressions in POOMA II are also first-class arrays.

POOMA hides the details of parallel computation in a flexible *evaluator* architecture. For the user, this means that a program can be written in a highly abstract data-parallel form, tested and debugged in serial mode, and then run in parallel with very little effort. In the future, POOMA will support cross-box parallelism using MPI, fields, particles, automatic boundary conditions, and flexible geometry and meshes.

The ACL website contains descriptions and contact information for several more projects at the ACL. Here are brief descriptions of some of them.

- PAWS (The Parallel Application Work Space) provides a framework for coupling parallel applications using high-speed parallel-communication channels. The coupled applications can be running on heterogeneous machines, and the data structures in each coupled component can have different parallel distributions.
- SILOON (Scripting Interface Languages for Object-Oriented Numerics) lets scientists rapidly prototype and solve problems on high-performance parallel computers. SILOON lets scientists and other application programmers easily access existing object-oriented scientific frameworks and numerical libraries written in C, C++, and Fortran.
- SMARTS (Shared Memory Asynchronous Runtime System) supports integrated task and data parallelism for MIMD ar-



chitectures with deep memory hierarchies.

- TAU (Tuning and Analysis Utilities) is a toolset that lets users analyze the performance of parallel-application programs. TAU collects much more information than is available through *prof* or *gprof*, the standard Unix utilities.

The Python Journal

The Python Journal is online at <http://www.pythonjournal.com/>.

The first issue featured an interview by Paul Everitt of Infoseek, a report on XML by Andrew Kuchling, and an article on Python as a rapid prototyping language by Sean Reifschneider. Regular features will include an Algorithms column by Andrew Kuchling, and a "From the Trenches" column by Sean Reifschneider. The Python website is <http://www.python.org>.

Numerical linear algebra for high-performance computers

Jack Dongarra, Iain Duff, Danny Sorensen, and Henk van der Vorst have a new book out, *Numerical Linear Algebra for High-Performance Computers* (SIAM, ISBN 0-89871-428-1). Topics include major elements of advanced-architecture computers and their performance, recent algorithmic development, and software for direct solution of dense matrix problems, direct solution of sparse systems of equations, iterative solution of sparse systems of equations, and solution of large sparse-eigenvalue problems.

Logically Interfacing C++ to Fortran 95

To logically interface C++ to Fortran 95, we want an object defined and implemented in one language to appear as

a natural object in the other language. Specifically, we want to

- Mirror C++ objects with Fortran 95 objects

- Mirror Fortran 95 objects with C++ objects

Fortran 95 shadow of C++ Objects. Consider the C++ class header

shown in Figure 2. To shadow this C++ class in Fortran 95, we must first export a flat interface that can negotiate the physical language barrier.

To flatten the interface, the implicit this pointer present in every member function call must be made explicit and representable by one of the allowable common types. One nonportable way of associating a pointer with an intrinsic type is to cast the pointer to a long, which can be passed as an integer to Fortran 95. We reject this option; we would like to keep the unavoidable portability issues isolated at the physical interface level.

A guiding principle of object-oriented thinking is that the solution to a problem lies in the creation of the appropriate class. Here the solution would be a class that associates an opaque pointer,^{8,9} a variable of an allowable type (the `opaque_pointer_type` could be, for example, long), with the actual object.

Figure 3 shows the the Design by Contract¹⁰ requirements for an opaque-pointer class. This specification is templated on the type; each class-member function is specified by its signature, allowable inputs (the `assert` statements commented with `REQUIRE`), and acceptable output (the `assert` statements commented with `ENSURE`). Our actual implementation uses the STL `map` class.

The C++ code in Figure 4 exports the member functions to a flat C++ interface. This physical interface

- uses `extern "C"` to ensure name unmangling; appends an underscore and uses lowercase in keeping with the particular C++ and Fortran 95 compiler requirements,
- uses a long opaque pointer managed by the `opaque_pointers<foo>` class to flatten the user-defined type to one of the allowable intrinsic types, and
- is compiled together with the C++

```
template <class T>
opaque_pointer_type opaque_pointers<T>::insert(T *t)
    // add t to list, return associated opaque pointer self
    assert(!opaque_pointers<T>::is_full()); // REQUIRE
    assert(t == opaque_pointers<T>::item(self)); // ENSURE

template <class T>
bool opaque_pointers<T>::is_full()
    // is there no more room?

template <class T>
T *opaque_pointers<T>::item(opaque_pointer_type self)
    // convert opaque pointer to real pointer
    assert(opaque_pointers<T>::has(self)); // ENSURE

template <class T>
bool opaque_pointers<T>::has(opaque_pointer_type self)
    // is self associated?

template <class T>
void opaque_pointers<T>::erase(opaque_pointer_type self)
    // remove pointer referenced by opaque pointer self
    assert(opaque_pointers<T>::has(self)); // REQUIRE
    assert(!opaque_pointers<T>::has(self)); // ENSURE
```

Figure 3. C++ opaque pointer specification. The opaque pointer class holds pointers to type `T` and associates each with an `opaque_pointer_type` index `self`. Any implementation that conforms to this specification is suitable for flattening the interface.

```
This file should be generated automagically from foo.hh

#include "foo.hh"
#include "opaque_pointers.hh" // opaque pointers template

extern "C" { // external interface to F95

void construct_foo_(long &self)
{
    foo *t = new foo(); // construct foo
    self = opaque_pointers<foo>::insert(t); // get opaque pointer
}

void set_foo_(long &self, int &i)
{
    foo *t = opaque_pointers<foo>::item(self); // get foo pointer
    t->set(i); // dispatch call
}

void get_foo_(long &self, int &i)
{
    foo *t = opaque_pointers<foo>::item(self); // get foo pointer
    i = t->get(); // dispatch call
}

void destruct_foo_(long &self)
{
    foo *t = opaque_pointers<foo>::item(self); // get foo pointer
    delete t; // destroy foo
    opaque_pointers<foo>::erase(self); // remove opaque pointer
}

} // prepended underscores
// for these particular
// compilers
```

Figure 4. C++ flat interface. Member functions of class `foo` are dispatched from C functions using only intrinsic types.


```

! This file should be generated automatically from foo.hh

module foo_class
! Definition of foo_class (the shadow class)
implicit none
private

public :: construct, get, set, destruct

type, public :: foo           ! F95 shadow class
  private
  integer :: this             ! opaque pointer to C++ object
end type foo

interface construct           ! build foo
  module procedure foo_construct
end interface

interface get                 ! return component data value
  module procedure foo_get
end interface

interface set                 ! set component data
  module procedure foo_set
end interface

interface destruct           ! destroy foo
  module procedure foo_destruct
end interface
! see following figure

```

Figure 5. Fortran 95 shadow class interface. The opaque pointer to the C++ object is stored in the `this` component..

```

! see previous figure
contains

  subroutine foo_construct(self)
    type(foo), intent(inout) :: self

    call construct_foo(self%this) ! construct C++ object
  end subroutine foo_construct

  subroutine foo_set(self, i)
    type(foo), intent(inout) :: self
    integer, intent(in) :: i

    call set_foo(self%this, i) ! dispatch this->set(i)
  end subroutine foo_set

  function foo_get(self) result(i)
    type(foo), intent(in) :: self
    integer :: i

    call get_foo(self%this, i) ! dispatch this->get()
  end function foo_get

  subroutine foo_destruct(self)
    type(foo), intent(inout) :: self

    call destruct_foo(self%this) ! destroy C++ object
  end subroutine foo_destruct

end module foo_class

```

Figure 6. Fortran 95 shadow class implementation. Module procedures of the Fortran 95 `foo` class use the opaque pointer `this` to dispatch commands through the flat interface to the member functions of the C++ `foo` class.

class code in a shared library for automatic initialization

This flat-interface representation of the C++ class is precisely that described by James Rumbaugh and his colleagues¹¹ for object-based programming in Fortran 77. The object identity is specified by the opaque pointer `self`; the address of the actual object is retrieved by indexing the `opaque_pointers<foo>` class; functionality is retrieved by dispatching member functions from that pointer. With this flat interface, the C++ class `foo` could be used from C++, Fortran 77, or any procedural language.

Of course, this interface is not type safe; any integer value could be used as the `self` argument, with disastrous results. Because our Fortran 95 classes are type safe, part of the desire for a logical interface is the recovery of type safety.

Because the flat interface follows the same format as our Fortran 95 class module-procedure interface,¹ it is thus eminently suitable for use with our Fortran 95 class structure.

The Fortran 95 code in Figures 5 and 6 uses this interface to create a shadow class. Here we first recover type safety by encapsulating the opaque pointer `this` inside a Fortran 95 user-defined type. Each flat-interface function is likewise encapsulated in a Fortran 95 module procedure. The `foo_class` module defines a Fortran 95 class that shadows the C++ `foo` class; variables of type `foo` in the Fortran 95 code use the identity, state, and behavior of a C++ `foo` object.

C++ shadow of Fortran 95 objects. Consider the Fortran 95 class shown in Figure 7. To shadow this Fortran 95 class in C++, we must first export a flat interface that can negotiate the physical language barrier.

To flatten the interface, the explicit `self` argument present in every module procedure must be converted to one of the allowable common types. Because standard Fortran 95 does not give the user access to object addresses, casting the address to an integer is not an option. We must use an opaque pointer class.

Figure 8 shows the the Design by Contract requirements for an opaque pointer class. This specification is templated on the type, `$1`; each subroutine is specified by its signature, allowable inputs (the `REQUIRE` statements) and acceptable output (the `ENSURE` statements). Our actual implementation uses the `m4` preprocessor to define a macro that produces working code based on a fixed array of pointers.

Given any reasonable implementation of these requirements, the Fortran 95 code in Figure 9 exports the module procedures to a flat Fortran 77 interface. This physical interface:

- Uses external procedures to ensure name unmangling
- Uses an integer opaque pointer managed by the `bar_opaque_pointers` class to flatten the user-defined type to one of the allowable intrinsic types
- Is compiled together with the Fortran 95 class code in a shared library for automatic initialization

This flat-interface representation of the Fortran 95 class mimics what was done for the flat C++ interface. The object identity is specified by the opaque pointer `this` argument; a real pointer to the actual object is retrieved from the `bar_opaque_pointer` class; functionality is retrieved by dispatching method procedures with that pointer as first argument. With this

```
module bar_class
  ! Definition of bar (the real class)
  implicit none
  private
  public :: construct, get, set, destruct

  type, public :: bar                ! F95 class
  private
    integer i                        ! component data
  end type bar

  interface construct
    module procedure bar_construct !(self)
    ! build bar
    ! type(bar), intent(inout) :: self
  end interface

  interface get
    module procedure bar_get !(self) result(i)
    ! return component data value
    ! type(bar), intent(inout) :: self
    ! integer :: i
  end interface

  interface set
    module procedure bar_set !(self, j)
    ! set component data
    ! type(bar), intent(inout) :: self
    ! integer :: j
  end interface

  interface destruct
    module procedure bar_destruct !(self)
    ! destroy bar
    ! type(bar), intent(inout) :: self
  end interface

  contains
    ! Implementation omitted

end module bar_class
```

Figure 7. Fortran 95 class interface. We wish to create a flat physical interface to this class's module procedures so they can be invoked from C++, and a logical interface to this class's objects so they can be shadowed in C++.

```
function $1_opaque_pointers_insert(x) result(this)
  ! add x to list, return associated opaque pointer this
  type($1), pointer :: x
  integer :: this
  REQUIRE(.not. $1_opaque_pointers_full())
  ENSURE(x == $1_opaque_pointers_item(this))

function $1_opaque_pointers_full() result(l)
  ! is there no more room?
  logical :: l

function $1_opaque_pointers_item(this) result(s)
  ! convert opaque pointer to real pointer
  integer, intent(in) :: this
  type($1), pointer :: s
  REQUIRE($1_opaque_pointers_has(this))

function $1_opaque_pointers_has(this) result(l)
  ! is this associated?
  integer, intent(in) :: this
  logical :: l

subroutine $1_opaque_pointers_erase(this)
  ! remove pointer referenced by opaque pointer this
  integer, intent(in) :: this
  REQUIRE($1_opaque_pointers_has(this))
  ENSURE(.not. $1_opaque_pointers_has(this))
```

Figure 8. Fortran 95 opaque pointer specification. The opaque pointer class holds pointers to type `$1` and associates each with an integer index `this`. Any implementation that conforms to this specification is suitable for flattening the interface.

```

! This class should be generated automagically from bar_class.f95

INCLUDE([opaque_pointer_class.fm4])      ! m4 macros to make a
OP([bar])                                ! bar_opaque_pointers class

subroutine bar_construct(this)
  use bar_class
  use bar_opaque_pointers_class
  integer, intent(inout) :: this
  type(bar), pointer :: t
  allocate(t); call construct(t)          ! construct bar
  this = bar_opaque_pointers_insert(t)    ! get opaque pointer
end subroutine bar_construct

subroutine bar_get(this, i)
  use bar_class
  use bar_opaque_pointers_class
  integer, intent(in) :: this
  integer, intent(out) :: i
  type(bar), pointer :: s
  s => bar_opaque_pointers_item(this)     ! get bar pointer
  i = get(s)                              ! dispatch call
end subroutine bar_get

subroutine bar_set(this, j)
  use bar_class
  use bar_opaque_pointers_class
  integer, intent(inout) :: this
  integer, intent(in) :: j
  type(bar), pointer :: s
  s => bar_opaque_pointers_item(this)     ! get bar pointer
  call set(s, j)                          ! dispatch call
end subroutine bar_set

subroutine bar_destruct(this)
  use bar_class
  use bar_opaque_pointers_class
  integer, intent(inout) :: this
  type(bar), pointer :: t
  t => bar_opaque_pointers_item(this)     ! get bar pointer
  call destruct(t); deallocate(t)         ! destroy bar
  call bar_opaque_pointers_erase(this)    ! remove opaque pointer
end subroutine bar_destruct

```

Figure 9. Fortran 95 flat interface. Module procedures of class `bar` are dispatched from Fortran 77 functions using only intrinsic types.

```

// this file should be generated automagically from bar_class.f95

// Declaration of bar (the shadow class)

#ifdef BAR_HH
#define BAR_HH

class bar                                // C++ shadow class
{
public:
  bar();                                // build bar
  int get();                             // return member data value
  void set(int j);                       // set member data
  ~bar();                                // destroy bar
private:
  long self;                             // opaque pointer to F95 object
};

#endif

```

Figure 10. C++ shadow class header. The opaque pointer to the Fortran 95 object is stored in the `self` member data.

flat interface the Fortran 95 class `bar` could be used from Fortran 77, C++, or any procedural language.

As with the C++ flat interface, this Fortran 95 interface is not type safe; as with the C++ flat interface we will pro-


vide a shadow class to recover type safety in the logical interface.

For this flat interface we can write the C++ shadow class header in Figure 10 with the implementation shown in Figure 11.

Here we first recover type safety by encapsulating the opaque pointer `self` inside a C++ class. Each function of the flat interface is likewise encapsulated in a C++ member function. The `bar` class defines a C++ class that shadows the Fortran 95 `bar` class; variables of type `bar` in the C++ code use the identity, state, and behavior of a Fortran 95 `bar` object.

We have presented techniques for shadowing C++ classes in Fortran 95 and vice versa. These techniques permit the export of classes in one language to the other, without sacrificing abstraction or type safety. Furthermore, these techniques are highly automatable; a script could easily read the C++ class header and construct the code necessary for the Fortran 95 shadow class, or read the Fortran 95 class module and construct the C++ shadow class.

In the process, we have created flat interfaces to objects in both languages that can be used for Fortran 77, C++, or any similar language.

These shadow ideas can be extended to any argument types as well; ultimately, we see no reason why entire component libraries cannot be shadowed. This permits a programmer to use the best programming language for the task at hand. 

FILL
GOES
HERE

```
// This file should be generated automagically from bar_class.f95

// Definition of bar (the shadow class)

#include "bar.hh"

extern "C" {                                     //external interface to F95

void bar_construct(long &self);

void bar_set(long &self, int &i);

void bar_get(long &self, int &i);

void bar_destruct(long &self);

}                                                // prepended underscores for
                                                // these particular compilers

bar::bar()
{
    bar_construct_(self);                        // construct f95 object,
                                                // return opaque pointer
}

int bar::get()
{
    int i;

    bar_get_(self, i);                          // dispatch i = get(self)
    return i;
}

void bar::set(int j)
{
    bar_set_(self, j);                          // dispatch call set(self, j)
}

bar::bar()
{bar_destruct_(self);                          // destroy f95 object}
```

Figure 11. C++ shadow class implementation. Member functions of the C++ bar class use the opaque pointer *self* to dispatch commands through the flat interface to the module procedures of the Fortran 95 bar class.

References

1. M.G. Gray and R.M. Roberts, "Object-Based Programming in Fortran 90," *Computers in Physics*, Vol. 11, No. 4, 1997, p. 355.
2. A.D. Robison, "C++ Gets Faster for Scientific Computing," *Computers in Physics*, Vol. 10, No. 5, 1996, p. 458.
3. G. Furnish, Disambiguated Glommable Expression Templates," *Computers in Physics*, Vol. 11, No. 3, 1997, p. 263
4. M.J. LeBrun, G. Furnish, and T. Richardson, *The PLPLOT Programmer's Reference Manual*, IFSR No. 538, Univ. of Texas, Austin, Tex, 1994.
5. B.D. Burow, "Mixed Language Programming," *Conf. High-Energy Physics CHEP95*, 18–22 Sept. 1995; http://www.hep.net/conferences/chep95/html/abstract/abs_69.htm.
6. *International Standard Programming Languages—C++*, ISO/IEC 14882, Am. Nat'l Standards Inst., Information Technology Industry Council, New York, 1998, S 3.6.2 and S 6.7 P 4.
7. *Solaris 2.6 Software Developer Collection Vol. 1—Linker and Libraries Guide*, Sun Microsystems, 1997, <http://docs.sun.com>.
8. J. Lakos, *Large-Scale C Software Design*, Addison-Wesley, Reading, Mass., 1996.
9. J.O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1994.
10. *Design by Contract*, Interactive Software Engineering, ISE Building 270 Storke Rd., Goleta, Calif; <http://www.eiffel.com/>
11. J. Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice-Hall, New York, 1991.